Applying Software Reliability Engineering in the 90's

William Everett, Senior Member, IEEE

Software Process and Reliability Engineering, Albuquerque

Samuel Keene, Fellow, IEEE

Performance Technology, Boulder

Allen Nikora, Member, Reliability Society and Computer Society

Jet Propulsion Laboratory, Pasadena

Key Words - Reliability, Software Reliability, Reliability Modeling and Prediction, Software Reliability Tool Kits, Software Reliability Engineering (SRE)

Summary & Conclusions - This paper reviews the progress in software reliability over the past 15 years and discusses the best tools and practices that can be applied today. Software is seen to play an increasingly vital role over time, vis-a-vis hardware, in terms of system content. The software content is increasing and, often today, it is a key factor in safety critical applications in medicine, transportation, and nuclear energy. Significant software content is found in almost every system, appliance, and machine which we use. In addition, software is the backbone of our business enterprise operations.

Consequently, producing reliable software is a mandate. Its development is often the "long pole in the tent" driving the cycle time required to produce and field a product. Most often, software is the main source of system reliability problems. The best development practices are recommended, herein, for managing the reliability of software. The development of software reliability models and user-friendly tool kits is described. These tools allow software reliability to be measured, tracked, and improved to meet the customer's specified reliability. The ability to measure software reliability promotes development focus, and consequently, its improvement.

1.0 INTRODUCTION

1.1 Background

In 1984, Professor Martin Shooman wrote a benchmark article on the History of Software Reliability [23]. Our paper updates the developments in software reliability that have occurred since then. Koss notes that "software reliability has been so overlooked... As late as 1986-7 (some) major command and control systems had not made one assessment of software reliability"[14]. He pointed out that the software in modern warplanes exceeds a million lines of code. He further states that "the ability to deliver reliable computer hardware can be considered to be a given. It is the ability to deliver the software of the system which will determine the extent to which the total system meets its operational availability." All too often, software reliability continues to be neglected. Even today, too few organizations even measure software reliability and some of those who do only measure it from a historical perspective. To be proactive, software reliability should be managed throughout the development process, starting from the requirements definition phase. This paper addresses some worthwhile initiatives to better manage and measure software reliability.

Software reliability is the dominant driver of today's system reliability. Software driven outages have been reported to exceed hardware outages by an order of magnitude [13]. Murphy points out that the main driver of field problems on over 2,000 European deployed Digital Equipment Corporation Systems fall into the class of System Management failures [20]. These problems are due to requirements and interface deficiencies. The present authors recognize these problem causes as a subset of software faults.

It should be noted that some noted reliability practitioners still question whether software can indeed fail. They argue that nothing actually "breaks" since its physical state remains unchanged-thus, "no broken" code exists. What fails is the software's ability to perform its intended or desired function, forcing the customer—the final arbitrator—to declare failure.

Software is proliferating in our every day products. Software is also embedded in machine logic in the form of "firmware". Firmware is software that resides in a non-volatile medium that is read-only in nature and is write-protected when functioning in its operational environment. It cannot be modified during program execution. Many common products have significant firmware content. Automobiles, telephone routers, and some appliances incorporate up to 1,000,000 bytes of stored firmware. Firmware promotes personalization of the equipment's functionality so it can be customized to fit different application needs. This firmware capability also provides control and diagnostic information. This design flexibility and extra functionality can lead to reliability and safety problems.

1.2. Notable Reliability and Safety Problems

Unfortunately, software is not built with the same degree of provable components found in hardware. It is also not tested as exhaustively as hardware and thus tends to have more residual design problems. Commercial software products typically are shipped with one or more defects per KSLOC. One defect per KSLOC would be considered a relatively good latent defect level. This means that a 1,000 KSLOC of code, contains 1,000 latent defects at shipment. These defects will be left to the customer to potentially experience before they are removed. To some hardware designers, this defect level would seem excessively high to their standards.

Software has caused some high profile reliability and safety problems. Neumann produced an excellent synopsis of many of these problems [21]. The Neumann web page also publicizes current software and system problems.

For example, the Therac 25 therapy accelerator irradiates tumors in two modes: electron beam bombardment and X-ray mode. The first therapy mode is low energy bombardment. The second mode Intersperses a Tungsten target into the electron beam before raising the beam energy one hundred fold. This puts the Therac 25 into the X-ray therapy mode. When "malfunction 54" appeared on the operator's screen, the system's operating modes became scrambled--exposing the patients with a lethal level of electron beam radiation. "Malfunction 54" has killed two people and harmed several more [9].

Therac 6, the predecessor irradiation model used mechanical interlocks rather than software interlocks. It never experienced this safety critical failure mode. .

Another example is the Patriot missile system. The United States used this weapon against the Iraqi's Scud missiles in the Persian Gulf war. During this war, they discovered a deadly design flaw: the Patriot missile had an imprecise timing calculation. This calculation appears to be responsible for the missile's failure to detect and engage the Scud missile that killed and injured troops in Tehran [22]. The original operational profile for the Patriot missile required it being relocated twice a day. This was necessary to keep a sophisticated enemy from tracking the Patriot's location from its own radar emissions. The Patriot's software was re-initialized every time it moved. This reset the accumulating timing error to zero twice a day. Because Iran was not considered a sophisticated enemy, the Patriots were left in place for long periods of time. The timing errors accumulated to the point of defeating the missile's ability to intercept Scud missiles, resulting in 126 casualties.

Another subtle design problem was subsequently found with the Patriot's software. Two different and unequal versions of the number 0.1 were implemented in 24-bit and 48-bit representations [21]. This led to a residual comparative error when there should have been none.

The most widely discussed reliability problem, in history, is the year 2000 or Y2K problem. Caper Jones estimated that the total industry cost to repair this one type defect is $276 billion US [10]. The problem is that so much of our legacy code is programmed in two digits instead of four digits. Consequently, these defective program dates will roll over at the turn of the century and the system will believe the date is 1900. Voas states to understand the magnitude of the problem consider the grocery store checkout lane [24]. Ever been in a line when the grocery scanner is not working or the credit card reader is not working? The clerk and the customer are quickly frustrated. Pandemonium reigns. Now imagine that this same experience is happening in offices, businesses, and stores everywhere. The impact could be disastrous--and possibly lethal. Some analysts worry that Y2K could send the economy into a recession as the cost of the problem rivals the

annual budget for software production [18]. Clearly, the best development practices and tools are needed to thwart such problems from occurring.

2.0 Software Reliability Engineering

2.1 Development of Best Practices

Nearly 31 years have transpired since Hudson's first significant study of software reliability [4]. Most of these early studies were focused on applying reliability growth models to failures data collected during the testing or field operation of software products. By the late 1980's, between 50 and 100 models surfaced for software reliability. A good summary of the history of these earlier developments is provided in [6]. The number of these models reflected an active and healthy period of research.

Although sufficient models existed to analyze the reliability of software, no guidelines and practices were available to help practitioners apply them to their products. Moreover, the shear number of models only added to their confusion. To address this problem, the American Institute of Aeronautics and Astronautics established a Blue Ribbon panel. The panel recommended four reliability growth models and provided procedures for collecting reliability data to use with these models [1]. The models recommended were:

- The Schneidewind Model.
- The Jelinski/Moranda Model.
- Musa/Okumoto Logarithmic Poisson Execution Time Model.
- The Littlewood Verrall Model.

The AIAA guidebook recommends that projects apply several models to estimate reliability and compare results by using each model. Several factors are suggested for comparing model results including:

- how valid are model predictions?
- how easy is it to acquire measurement data to use the model?
- how close are models assumptions met by situation being modeled?
- can the model estimate useful quantities needed by project personnel?
- how robust is the model to changes in the test and operational environment?
- is the model and its use simple to understand?
- is the model insensitive to noise?

Around 1988, efforts were initiated to encourage more active use of software reliability methods by practitioners. At that time, AT&T Bell Laboratories introduced a sequence of internal courses around the notion of software reliability engineering (SRE). The definition of SRE went beyond modeling and measuring the reliability of software to include the application of these models and measures to managing the reliability of software. In 1990, they held a kick-off meeting of the IEEE Subcommittee on Software Reliability Engineering. This was the start of an annual series of international symposiums on SRE (ISSRE). The eighth annual symposium, ISSRE '97, was just held in Albuquerque New Mexico. These symposia not only provided forums for communication among researchers, but also among practitioners. Over 60% of the attendees of ISSRE '97 were from industry or government. Although early ISSREs had a strong focus on modeling methods, more varied topics have been covered in later ISSREs. ISSRE '97 included sessions on *Fault-Prone Module Identification, Error Detection and Handling, Test Strategies, Software Process Effectiveness, and Process and Quality*--plus a number of industry practice sessions. (See reference [7] .) A case studies handbook [8] highlighting particular industrial and practical applications of SRE was also published as part of ISSRE '97.

2.2 SRE Institutionalized

In 1992, AT&T Bell Laboratories adopted a best current practice for doing SRE. (A condensed version can be found in [3]). This practice defined 25 activities that should be included in a good SRE program. (See Table 1.) As Table 1 shows, the practice

covers the entire life cycle of product development--from the earliest stages of product conceptualization through post delivery support of the product.

Life Cycle Phase	SRE Activities
Feasibility And Requirements	• Determine functional profile • Define and classify failures • Identify customer reliability needs • Conduct trade-off studies • Set reliability objectives
Design And Implementation	• Allocate reliability among components • Engineer to meet reliability objectives • Focus resources based on functional profile • Manage fault introduction and propagation • Measure reliability of acquired software
System Test And Field Trial	• Determine operational profile • Conduct reliability growth testing • Track testing progress • Project additional testing needed • Certify reliability objectives are met
Post Delivery And Maintenance	• Project post-release staff needs • Monitor field reliability versus objectives • Track customer satisfaction with reliability • Time new feature introduction • Guide product and process improvement

Table 1. SRE life cycle activities

2.3 SRE Activities

Activities during the *Feasibility and Requirements* phase focus on establishing reliability requirements for the product. It starts with the user of the product and includes defining what types of failures the potential user may experience in using the product and

how costly these failures would be to the user in the work that they do. The likelihood of failures in a software product depends heavily on how the user employs the product. During the early stages of product development, this usage is captured in something called a functional profile. The functional profile describes high level functions performed by the user and how often these functions are performed. Trade-offs need to be made in establishing overall product requirements. Although adding more features into a software product will make it more desirable to a user, it will also add to the development costs--especially to those costs related to managing product reliability.

From a reliability standpoint, the outcome of the *Feasibility and Requirements* phase is a set of reliability requirements. Properly defining system and software requirements is most important since requirement deficiencies are typically the number one cause of problems with field maintenance [12]. These requirements must not only specify objectives by failure class, but also the conditions under which the objectives are to be met. Understanding them is an evolutionary process that works best when they are the collaborative effort of both the developer and the customer.

Activities during the *Design and Implementation* phase are focused on developing a product that meets reliability objectives. The first activity is to allocate overall reliability among the components of the product. A number of steps can be taken to engineer the product so it meets its reliability objectives. A number of these are a carryover from hardware and system reliability methods. They include techniques such as fault tree analysis (FTA) and failure mode and effects analysis (FMEA). Since software failures are caused by residual faults in their design and development process, reliability methods must target processes that introduce faults and allow them to propagate to later development phases.

In addition, today's software products may include new or reused components developed (by other organizations). The reliability of this *acquired* software must also be managed.

Activities during the *System Test and Field Trial* phase are targeted at validating the developed software so it meets its reliability objectives. In product testing, we must

mimic how the customer will use the software if we want an accurate view of the product's reliability.

The *functional profile* developed during the *Feasibility and Requirements* phase provides a start in specifying how the user will use the product. With the product designed and developed, we redefine the functional profile in terms of operations offered by the system to perform specific functions. For example, in an Automated Teller Machine (ATM) product, a function would be for a user to deposit money while the operations would be the ATM screens the user would need to do the deposit. The result would be an *operational profile.* Again, the *operational profile* will be used in developing test cases and in specifying the frequency and order in which they will be run. Failure data is collected during testing and is used to calibrate a reliability growth model. The calibrated model is used in turn to tell us the current level of reliability of the software product and can even be used (with adequate collected data) to estimate the remaining testing time needed to reach a reliability objective. Reliability growth is experienced during testing because once a failure is encountered, the underlying fault that triggered the failure is removed and thus will never cause another like failure.

The management of reliability does not stop when the product is delivered to the user, it continues during the *Post Delivery and Maintenance* phase. One activity focuses on estimating the amount of staff needed to provide hot-line support to help users with field-reported failures and staff needed to fix the software product. Field reliability of the software product should still be tracked to ensure the user is satisfied with product reliability. As fixes are introduced into the fielded software product, reliability will continue to grow. We should time the release of new software with major feature enhancements at points where the reliability level perceived by the user continues to be acceptable. Finally, we will use the information we gathered during this phase to improve our development processes that impact reliability and to improve the reliability of subsequent releases of the software product or new products.

3.0 RELIABILITY MODELING

3.1 Modeling Background

Another important step in making SRE techniques more accessible to practitioners was the development of tools. Although a large number of software reliability models have been published since the first models were published in 1971, it is only since the mid-1980s that tools implementing these models have become widely available. Prior to their advent, development organizations wishing to use software reliability modeling techniques to monitor and control their development efforts had little choice but to develop their own tools. Because of the computational complexity of the models, the development of a software reliability modeling tool is a significant effort in its own right--and one to which many organizations were not willing to devote resources that could be applied to producing commercial systems. The advent of widely available tools could then be considered as important as increasing interest in the use of software reliability measurement.

One of the first software reliability measurement tools was developed by AT&T in 1977. Although originally intended for in-house use, it has been commercially available for the past ten years. The tool implements two models: the Musa Basic and Musa/Okumoto logarithmic Poisson models. The tool outputs can easily be related back to the development process. Rather than simply providing estimates of the model parameters, the tool provides estimates of initial current failure intensities as well as confidence intervals around these estimates. In addition, it predicts the amount of time required achieving user-specified failure intensity as well as the number of additional failures that will be seen before the specified failure intensity is achieved. This tool takes both time-domain and interval-domain failure data as input. Outputs are shown in both tabular and graphical fashion. Besides the outputs mentioned above, other plots are available which allow users to see how well the model results fit the data, to see trends in the initial and current failure rates, and to see predictions of a development effort's completion date.

3.2 SMERFS Tool

One of the next major achievements was the development of SMERFS (Statistical Modeling and Estimation of Reliability Functions for Software) at the Naval Surface

Warfare Center in Dahlgren, VA. First released in 1983, this was the first tool to implement a wide variety of software reliability models. It included interval-domain (e.g., Schneidewind, Yamada S-Shaped, Brooks and Motley) as well as time-domain (Musa Basic, Musa/Okumoto, Littlewood-Verrall) models. This last can be of particular interest to development organizations, since failure history data tends to be more widely available as the number of failures observed per test interval of a given length rather than as interfailure times. SMERFS was designed for ease-of-use. It has a menu-driven interface, which partitions the functionality into well-defined areas. These areas are data entry, editing, and transformation; model application; and determination of model applicability. Users can specify whether model parameters should be made using maximum-likelihood estimation or least squares. Model results are displayed in an easy to read tabular form and always include estimates of the model parameters. In addition, each model has its own specific set of results. They include the following: expected time to next failure, estimated total number of failures, estimate of the reliability for a specified time, number of failures remaining in the system, and the expected number of failures in a session of a specified duration. The sole model evaluation criterion for the earlier versions was goodness-of-fit (Chi-Square test for interval-domain data and 2-tail Kolmogorov-Smirnov test for time-domain data). However, the current version also includes prequential likelihood, model bias, bias trend, and noise [2]. SMERFS was also designed to allow users to extend its capabilities. Unlike other software reliability modeling tools, SMERFS is distributed with the source code, which is a subset of ANSI FORTRAN '77, as well as design documentation. This allows users to customize the user interface to meet their own needs, as well as add models of their own.

3.3 Other Tools

In 1988, Reliability and Statistical Consultants, Ltd. Developed the Software Reliability Modeling Program (SRMP) in the United Kingdom Ltd. The distinguishing feature of this program is its ability to include statistical methods other than goodness-of-fit for identifying the most appropriate model for a set of failure history data. These techniques have become an essential part of an analyst's tool kit. In 1986, Abdel-Ghaly

et al [2] had shown that it does not seem possible to select a priori the most appropriate model for a development effort. Rather, a set of models should be run against a set of failure data and the results analyzed to identify the most appropriate model. Abdel-Ghaly and his associates developed a set of statistical methods for identifying:

- How much more likely it is that one model will produce more accurate predictions than another model. To compare two models, A and B, we first compute the prequential likelihood functions for each model, PL_A and PL_B. The ratio PL_A / PL_B specifies how much more likely it is that model A will produce accurate estimates than model B, given that there is no preference for either model prior to applying them.
- The tendency for the model to produce biased results. For instance, a model may consistently predict interfailure times shorter than those actually observed. The technique for determining whether a model is biased is known as a u-plot.
- The tendency for the model bias to shift with time. It may be that in the early stages of a testing effort, a model will be optimistically biased (predicting interfailure times that are shorter than those actually observed). During later stages of testing, the model may assume a pessimistic bias (predicting longer interfailure times than those actually observed). The technique for determining whether a model exhibits temporal shifts in its bias is known as a y-plot.

SRMP computes the prequential likelihood, u-plot, and y-plot for each of the nine models it implements, allowing users to determine the most appropriate model for their development effort. Unlike SMERFS, SRMP does not include traditional goodness-of-fit criteria such as the Chi-Square or Kolmogorov-Smirnov tests, and implements only time-domain models.

A distinguishing feature of the SoRel tool that was developed by LAAS, a laboratory of the National Center for Scientific Research in Toulouse, France, is a set of tests that can be applied to a set of failure data prior to model application to identify trends in the failure data. As a software system undergoes test, there may very well be times when the system is not experiencing reliability growth. For instance, the testing staff may be producing tests with the specific intention of revealing faults. Or, as new

functionality is added to the system, the testing team will focus on the new functionality, rather than sampling the operational profile. In either one of these cases, the failure data may appear to exhibit no improvement in reliability or an actual reliability decrease. The tests implemented by SoRel to identify these trends are the arithmetical test (e.g., a running average of interfailure times), the Laplace test [11], the Spearman test, and the Kendall test. This allows an analyst to determine whether a data set exhibits reliability growth, reliability decrease, or determine that there is no identifiable trend. An appropriate set of models can then be selected on the basis of these trend tests. None of the other tools described above offer this capability. Like SMERFS and SRMP, SoRel uses a variety of reliability models for either time-domain or interval-domain data. It also computes goodness-of-fit, prequential likelihood, and the residuals for each model that is run. SoRel runs on the Macintosh and is the only tool described so far to produce high-resolution plots of model results, although it requires Excel to do its plotting.

3.4 CASRE Tool

In 1992, Computer Aided Software Reliability Estimation (CASRE) was developed in response to a perception that many of the available software reliability tools were not easy for non-specialists to use. The developers of CASRE wanted to provide a system, suitable for use by both research and practitioners with the following characteristics:

- Allow users to select from a wide variety of time-domain and interval-domain models.
- Display model results as high-resolution plots and in tabular form.
- Guide users through the selection, execution, and evaluation of models through an appropriate set of structured menus.
- Minimize the amount of time required for users to learn how the tool, and minimize the amount of time required to re-learn the tool after having not used it for an extended interval.

Finally, one feature of CASRE drew on research indicating that one way of increasing the predictive accuracy of software reliability models was to form weighted sums of the

results of several models [15, 16]. In particular, linear combinations of model results in which the weights of each component of the combination are determined by comparisons of the change in the prequential likelihood values of each model over the past few observations appeared to provide the greatest increase in predictive accuracy [16]. The distinguishing feature of CASRE is that it allows users to form linear combinations of models according to one of three weighting schemes. CASRE runs in Windows 3.1, Windows95, or WindowsNT. It uses the SMERF's libraries to do the model computations and evaluations, taking advantage of existing software known to be suitable for the intended application and with an extensive history of use. The design of the user interface was guided by interviews with potential users to help meet the criteria listed above.

4.0 LAST THOUGHTS

Sometimes criticisms are heard that too many models of software reliability are a sign that no one knows what is going on. The plethora of models can optimistically be seen as evidence of the energy and excitement in the field. Obviously, extensive on-going research and vitality exist. It is also worth citing again the recent Handbook on Software Reliability. This is a very complete reference and would be a fundamental library addition to anyone interested in exploring this field. The text also contains a CD that has the reliability tools mentioned in this article.

REFERENCES

[1] AIAA, "Software Reliability Estimation and Prediction Handbook", 1992.

[2] A. Abdel-Ghaly, P. Chan, and B. Littlewood, "Evaluation of Competing Software Reliability Predictions," IEEE Transactions on Software Engineering, vol. SE-12, Sep. 1986, pp. 950-967.

[3] M. Donnelly, W. Bill Everett, J. Musa and G. Wilson, *Best Current Practice of SRE* in Handbook of Software Reliability Engineering, McGraw-Hill, 1996.

[4] G. Hudson, "Program Errors as a Birth and Death Process", System Development Corporation Report SP-3011, Santa Monica, Calif. 1967

[5] A. Iannino, and J. Musa, and *Software Reliability Engineering at AT&T* in Probabilistic Safety Assessment and Management, Elsevier Science Publishing Co., Inc., 1991.

[6] A. Iannino, *Software Reliability Theory* in Encyclopedia of Software Engineering (J. J. Marichiniak – editor) Wiley-Interscience, 1994.

[7] IEEE Computer Society, Proceedings of the Eighth International Symposium on Software Reliability Engineering, IEEE Computer Society Press, November 1997.

[8] IEEE Computer Society, Software Reliability Engineering Case Studies, IEEE Computer Society Press, November 1997.

[9] E. Joyce, "Software Bugs: A Matter of Life and Liability", Datamation Magazine, May 15, 1987, pp. 88-92.

[10] C. Jones, "Year 2000: What's the Real Cost?", Datamation, 1997 March, pp. 88-93.

[11] K. Kanoun, M. Bastos Martini, J. Moreira De Souza, "A Method for Software Reliability Analysis and Prediction – Application to the

TROPICO-R Switching System,", IEEE Transactions on Software Engineering, April 1991, pp. 334-344.

[12] S. Keene, T. Keller, and J. Musa, "Developing Reliable Software in the Shortest Duty Cycle," IEEE Video Tutorial Tape, ISBN 0-7803-2850-7, IEEE, Piscataway, N.J., (1-800-678-IEEE)

[13] S. Keene, "Modeling Software R&M Characteristics", ASQC Reliability Review, Parts I and II, Vol. 17, Nos. 2 & 3, 1997 June, pp. 5-28, 1997 Sept., pp. 13-22.

[14] E. Koss, "Software Reliability Metrics for Military Systems", 1988 Proc Ann. Reliability and Maintainability Symposium, 1988, pp. 190-194.

[15] M. R. Lyu and A. P. Nikora, "A Heuristic Approach for Software Reliability Prediction: The Equally-Weighted Linear Combination Model," published in the proceedings of the IEEE International Symposium on Software Reliability Engineering, May 17-18, 1991, Austin, TX

[16] M. Lyu and A. Nikora, "Applying Reliability Models More Effectively", IEEE Software, vol. 9, no. 4, pp. 43-52, July

[17] M. Lyu, (Editor), Handbook of Software Reliability Engineering, McGraw-Hill, 1996.

[18] M. Mandel, P. Coy, and C. Judge, "Zap! How the Year 2000 Bug will Hurt the Economy", Business Week, 1998 March 2, pp. 93-97.

[19] J. Musa, A. Iannino, and K. Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, 1987.

[20] B. Murphy and T. Gent, "Measuring System and Software Reliability Using an Automated Data Collection Process", Quality and Reliability Engineering International. 1995 xxxx

[21] P. Neumann, "Computer Related Risks", Addison-Wiley, New York, 388 pages, p.34.

[22] M.Riezenman, Ed., "Revising the Script After Patriot", IEEE Spectrum, 1991 September, pp. 49-52.

[23] M. Shooman, "Software Reliability: A Historical Perspective", IEEE

Trans. Reliability, Vol. R-33, No. 1, 1984 April, pp. 48-55.

[24] J. Voas and G. McGraw, "Software Fault Injection", John Wiley & Sons, Inc., New York, 1998, 354 pages.

Cooresponding author: Samuel Keene

650 Tamarack Ave 3810

Brea, Ca 92821

FAX 714-446-3132

Phone 714-446-3080